Building Natural Language Interfaces to Web APIs

Yu Su[1], Ahmed Hassan Awadallah[2], Madian Khabsa[2]
Patrick Pantel[2], Michael Gamon[2], Mark Encarnacion[2]
[1]University of California, Santa Barbara, [2]Microsoft Research
ysu@cs.ucsb.edu,{hassanam,makhab,ppantel,mgamon,markenc}@microsoft.com

ABSTRACT

As the Web evolves towards a service-oriented architecture, application program interfaces (APIs) are becoming an increasingly important way to provide access to data, services, and devices. We study the problem of natural language interface to APIs (NL2APIs), with a focus on web APIs for web services. Such NL2APIs have many potential benefits, for example, facilitating the integration of web services into virtual assistants.

We propose the first *end-to-end* framework to build an NL2API for a given web API. A key challenge is to collect training data, i.e., NL command-API call pairs, from which an NL2API can learn the semantic mapping from ambiguous, informal NL commands to formal API calls. We propose a novel approach to collect training data for NL2API via crowdsourcing, where crowd workers are employed to generate diversified NL commands. We optimize the crowdsourcing process to further reduce the cost. More specifically, we propose a novel hierarchical probabilistic model for the crowdsourcing process, which guides us to allocate budget to those API calls that have a high value for training NL2APIs. We apply our framework to real-world APIs, and show that it can collect high-quality training data at a low cost, and build NL2APIs with good performance from scratch. We also show that our modeling of the crowdsourcing process can improve its effectiveness, such that the training data collected via our approach leads to better performance of NL2APIs than a strong baseline.

KEYWORDS

Natural language interface; Web API; Crowdsourcing; Hierarchical Probabilistic Model

1 INTRODUCTION

Benefiting from a confluence of factors, such as service-oriented architecture (SOA), cloud computing, and Internet-of-Things (IoT), application program interfaces (APIs) are playing an increasingly important role in both the virtual and the physical world. For example, web services (e.g., weather, sports, and finance) hosted in the cloud provide data and services to end users via web APIs [1, 7], and IoT devices expose their functionalities via APIs to other devices on the network [9].

Ours
"How many unread emails about PhD application do I have?" ⇨ GET-Messages{(FILTER(isRead=False), SEARCH("PhD application"), COUNT()}
"Where is my next meeting?" ⇨ GET-Events{(SELECT(location),TOP(1), ORDERBY(start,asc),FILTER(start>now)}

IFTTT
"**If** I get tagged in a Facebook photo, **then** save it to Dropbox" ⇨ Trigger: Facebook.tagged_in_photo Action: Dropbox.save

Figure 1: NL command (left) and API call (right) pairs collected by our framework vs. IFTTT [19]. `GET-Messages` and `GET-Events` are different web APIs for searching emails and calendar events, respectively. An API can be called with different parameters. We target fully parameterized API calls, while IFTTT concerns API calls with simple parameters.

Traditionally APIs are mainly consumed by various kinds of software, e.g., desktop applications, websites, and mobile apps, which then serve users via graphical user interfaces (GUIs). GUI has made a great contribution to the popularization of computing, but many limitations gradually present themselves as the computing landscape evolves. On the one hand, as computing devices become smaller, more mobile, and more intelligent, the requirement of a screen for GUIs becomes a burden in many cases, e.g., for wearables and IoT devices. On the other hand, users have to adapt to different ad-hoc GUIs to use different services and devices. As the number of available services and devices rapidly increases, however, the learning and adaptation cost on users also increases. Natural language interface (NLI), also known as conversational user interface (CUI) and exemplified by virtual assistants like Apple Siri [3] and Microsoft Cortana [21], emerges as a promising alternative, which aims to act as a unified and intelligent gateway to a wide range of back-end services and devices.

In this paper, we study the problem of natural language interface to APIs (NL2APIs). Different from general-purpose NLIs like virtual assistants, we study how to build NLIs for *individual* web APIs, e.g., the API to a web service like ESPN for sports[1]. Such NL2APIs have the potential to address the *scalability* issue of general-purpose NLIs [21] by allowing for *distributed* development. The usefulness of a virtual assistant is largely determined by its breadth, i.e., the number of services it supports. However, it is very tedious for a virtual assistant to integrate web services one by one. If there is a low-cost way for individual web service providers to build an NLI to their respective APIs, the integration cost may be greatly reduced. A virtual assistant then needs not to handle the heterogeneous interfaces to different web services; rather, it only needs to integrate the individual NL2APIs, which enjoys the uniformity of natural

* The first author did this work mainly during an internship at Microsoft Research.

[1]Our core techniques can also be applied to other APIs like those for smartphone apps or IoT devices, but we focus on web APIs in this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM'17 , November 6–10, 2017, Singapore, Singapore
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4918-5/17/11...$15.00
https://doi.org/10.1145/3132847.3133009

language. On the other hand, NL2APIs can also facilitate web service discovery [20], recommendation [35], and help API programming by reducing the burden to memorize the available web APIs and their syntax [10].

Example 1. Two examples are shown in Figure 1. An API can be called with different parameters. For the email search API, users may filter emails by some properties or search for some keywords. Given an API, the main task of an NL2API is to map NL commands into the corresponding API calls.

Challenge. Training data collection is one of the most critical challenges of current NLI research and application [24, 28]. NLIs rely on supervised training data, consisting of NL command-API call pairs in the case of NL2API, to learn the semantic mapping from NL commands to the corresponding formal representations [32]. Due to the flexibility of natural language, users can describe an API call in syntactically divergent ways in natural language, i.e., *paraphrasing*. For the second example in Figure 1, users may also say "*Where is the upcoming meeting?*" or "*Find next meeting's location.*" It is crucial to collect sufficient training data in order to learn such language varieties [24]. Existing work of NLIs usually collect training data in a *best-effort* manner. For example, most related to ours is [19], which targets to map NL commands in IF-This-Then-That (IFTTT) form into API calls (Figure 1). Their training data is directly dumped from the IFTTT website (http://ifttt.com). However, if an API gets no or insufficient coverage, there is no way to remedy it. Also, training data collected in this way is hard to support advanced commands with multiple parameters. For example, we analyzed the annonymized logs of calls made to the Microsoft's email search API in one month, and found that roughly 90% involved 2 or 3 parameters, nearly evenly split, and the parameterizations are quite diversified. Therefore, we aim to fully exercise an API's parameterizations and support advanced NL commands. How to collect training data for a given API in such an active and configurable way remains an open problem.

Despite the extensive studies on NLI to other formal representations like relational databases [15], knowledge bases [4, 30], and web tables [18, 26], little research has been conducted on NLI to web APIs. We propose the first *end-to-end* framework to build an NLI for a web API *from scratch*. Given a web API, our framework consists of 3 steps: (1) *Representation.* The original HTTP format of web API contains many irrelevant and thus distractive details for NLIs. We propose an intermediate semantic representation for web APIs, so that an NLI can stay agnostic of the irrelevant details. (2) *Training data collection.* We propose a novel approach to solicit supervised training data from crowdsourcing. (3) *NL2API.* We propose two NL2API models, a language model based retrieval model [23] and a sequence-to-sequence (Seq2Seq) neural network model [27].

One of the key technical contributions of this work is a novel approach to actively collect training data for NL2API via crowdsourcing, where we employ humans to annotate API calls with NL commands. It has three design goals: (1) *Configurable.* One should be able to specify for which API, what parameterization, and how much training data she wants to collect. (2) *Low-cost.* Crowd workers, who are much more affordable than domain experts, should be employed. (3) *High-quality.* The quality of training data should not be compromised.

There are two main challenges in designing such an approach. First, API calls with advanced parameterizations, like the ones in Figure 1, are not understandable by average users, so *how to design the annotation task such that crowd workers can handle it with ease*? We start off by designing an intermediate semantic representation for web APIs (Section 2.2), which allows us to easily generate API calls with desired parameterizations. Then we design a grammar to automatically convert each API call into a *canonical* NL command, which may be somewhat clumsy but is understandable by average crowd workers (Section 3.1). Crowd workers only need to paraphrase the canonical command into a more natural way, which makes training data collection less error-prone because paraphrasing is a much easier task for crowd workers.

Second, *how to identify and only annotate the API calls of a high value for training NL2APIs*? Due to the combinatorial explosion of parameterization, the number of API calls, even for a single API, can be quite large. It is not economic, nor necessary, to annotate all of them. We propose a first-of-its-kind *hierarchical probabilistic model* for the crowdsourcing process (Section 3.2). Similar to language modeling for information retrieval [23], we assume NL commands are *generated* from the corresponding API calls, and estimate a language model for each API call to capture this generative process. The foundation of our model is the *compositional nature* of API calls, or that of formal meaning representations in general. Intuitively, if an API call is composed of some simpler API calls (e.g., "*unread emails about PhD application*" = "*unread emails*" + "*emails about PhD application*"), we can infer its language model from those of the simpler API calls, without even annotating it. Therefore, by just annotating a small number of API calls, we can estimate a language model for all the others. Certainly the estimated language models are not perfect, otherwise we would have already solved the NL2API problem. Despite the imperfect estimation, however, by foreseeing the language model of unannotated API calls, our model provides a holistic view of the whole API call space as well as the interplay of natural language and API calls, with which it becomes possible to optimize the crowdsourcing process. In Section 3.3, we present an algorithm that selectively annotates API calls with the objective to make different API calls more *distinguishable*, i.e., to make their language models more divergent from each other.

We apply our framework to two deployed web APIs from the Microsoft Graph API suite[2]. We demonstrate that high-quality training data can be collected at a low cost using the proposed approach[3]. We also show that our approach makes crowdsourcing more effective. Under the same budget, it can collect better training data than a strong baseline, leading to better accuracy of NL2APIs.

In summary, our main contributions are three-fold:

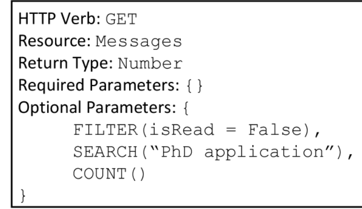
- We are among the first to study NL2API, and proposed an end-to-end framework to build an NL2API from scratch.
- We proposed a novel approach to collect training data for NL2API via crowdsourcing, and a novel crowdsourcing optimization strategy based on a first-of-its-kind hierarchical probabilistic model for the crowdsourcing process.
- We applied our framework to real-world web APIs and showed that reasonably performing NL2APIs can be built from scratch.

[2]https://developer.microsoft.com/en-us/graph/
[3]The dataset will be available at https://aka.ms/nl2api

Table 1: OData Query Options.

Query option	Description
`SEARCH(String)`	search for entities containing specific keywords
`FILTER(BoolExpr)`	filter entities according to certain criteria, e.g., `isRead=False`
`ORDERBY(Property, order)`	sort entities according to a property in 'asc' or 'desc' order
`SELECT(Property)`	instead of full entities, only return a certain property
`COUNT()`	count the number of matched entities
`Top(Integer)`	only return the first certain number of results

How many unread emails about PhD application do I have?



GET https://graph.microsoft.com/v1.0/<user-id>/ messages?$filter=isRead%20eq%20false& $search=PhD%20application&$count=true

Figure 2: API frame. Top: natural language command. Middle: API frame. Bottom: API call.

2 PRELIMINARY

2.1 RESTful API

In recent years, web APIs following the REST architectural style [1], i.e., *RESTful APIs*, are becoming more and more popular because of their simplicity. RESTful APIs are also used in smartphone apps [34] and IoT devices [9]. Restful APIs revolve around *resources*, addressable via URIs, and provide access to resources to a broad range of front-end consumers via simple HTTP verbs like `GET`, `PUT`, `POST`, etc. We will mainly work with RESTful APIs, but the core techniques can be generalized to other APIs.

We will adopt the popular Open Data Protocol (OData, [16]) for RESTful API, and will use two web APIs from the Microsoft Graph API suite as example (Figure 1), which are respectively used to search a user's emails and calendar events. In OData, resources are *entities*, each associated with a list of *properties*. For example, the `Message` entity, which represents email, has properties like `subject`, `from`, `isRead`, `receivedDateTime`, etc. In addition, OData defines a set of *query options* to enable advanced resource manipulation. For example, one can search for emails from a specific person or received on a certain date using the `FILTER` option. The query options we will use are listed in Table 1. We call each combination of an HTTP verb and an entity or entity set as an API, e.g., `GET-Messages` for email search. Each parameterized query option, e.g., `FILTER(isRead=False)`, is called a *parameter*, and an *API call* is an API with a list of parameters.

2.2 NL2API

The core task of an NLI is to map an utterance (natural language command) into a certain formal meaning representation, e.g., logical forms or SPARQL queries for knowledge bases [4], or web APIs in our setting. To better focus on the semantic mapping and stay agnostic from irrelevant details, an intermediate semantic representation is usually employed, instead of working directly with the target representation. For example, combinatory categorial grammar [32] has been widely used for NLI to data and knowledge bases. This kind of abstraction is also critical for NL2API. There are a lot of details, such as URL conventions, HTTP headers, and response codes, that can deviate an NL2API from the core semantic mapping task. Therefore, we define an intermediate representation for RESTful APIs (Figure 2), named *API frame* and reminiscent of frame semantics [2]. An API frame consists of five parts. *HTTP Verb* and *Resource* are basic in RESTful APIs. *Return Type* is useful for API composition, where we compose multiple API calls to fulfill a more complex task. *Required Parameters* are mostly used in `PUT` or `POST` API calls, e.g., sending an email requires recipient, title and body. *Optional Parameters* are often involved in `GET` API calls to specify detailed information need. When there are no required parameters, we serialize the API frame, e.g., `GET-messages{FILTER(isRead=False), SEARCH("PhD application"), COUNT()}`. An API frame can be converted into the real API call deterministically. Necessary contextual information, like user id, location, and datetime, will be filled in during the conversion. For the second example in Figure 1, the value `now` in the `FILTER` parameter will be replaced by the date-time of the specific command when converting the API frame to a real API call. We will use API frame and API call interchangeably hereafter.

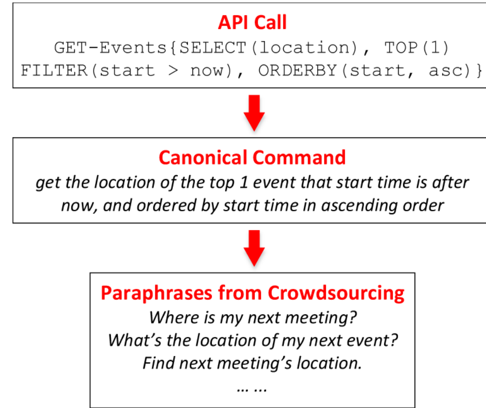


Figure 3: Crowdsourcing pipeline.

3 TRAINING DATA COLLECTION

In this section we propose a novel approach to collect training data for NL2API via crowdsourcing. We first generate API calls and convert each of them into a canonical command using a simple grammar (Section 3.1), and employ crowd workers to paraphrase the canonical commands (Figure 3). Based on the compositional nature of API calls, we propose a hierarchical probabilistic model for the crowdsourcing process (Section 3.2), and propose an algorithm to optimize the crowdsourcing process (Section 3.3).

<table>
<tr><td></td><td colspan="3">[Lexicon]</td></tr>
<tr><td>(L1)</td><td>get</td><td>→</td><td>V[GET]</td></tr>
<tr><td>(L2)</td><td>email</td><td>→</td><td>NP[Messages]</td></tr>
<tr><td>(L3)</td><td>sender</td><td>→</td><td>NP[from]</td></tr>
<tr><td>(L4)</td><td>in the category of</td><td>→</td><td>PP/NP[categories]</td></tr>
<tr><td>(L5)</td><td>read</td><td>→</td><td>JJ[isRead]</td></tr>
<tr><td>(L6)</td><td>have attachment</td><td>→</td><td>VP[hasAttachments]</td></tr>
<tr><td>(L7)</td><td>receive time</td><td>→</td><td>NP[receivedDateTime]</td></tr>
<tr><td>...</td><td></td><td>...</td><td></td></tr>
<tr><td></td><td colspan="3">[Boolean Expression]</td></tr>
<tr><td>(B1)</td><td>VP[x]</td><td>→</td><td>VP[x = True]</td></tr>
<tr><td>(B2)</td><td>do not VP[x]</td><td>→</td><td>VP[x = False]</td></tr>
<tr><td>(B3)</td><td>is JJ[x]</td><td>→</td><td>VP[x = True]</td></tr>
<tr><td>(B4)</td><td>is not JJ[x]</td><td>→</td><td>VP[x = False]</td></tr>
<tr><td>(B5)</td><td>is PP/NP[x] NP[y]</td><td>→</td><td>VP[x = y]</td></tr>
<tr><td>(B6)</td><td>NP[x] is before|after DATETIME[y]</td><td>→</td><td>NP[x <|> y]</td></tr>
<tr><td>(B7)</td><td>NP[x] is|is smaller than|is larger than *[y]</td><td>→</td><td>NP[x =|<|> y]</td></tr>
<tr><td></td><td colspan="3">[Query Option]</td></tr>
<tr><td>(Q1)</td><td>CP[x]</td><td>→</td><td>CP[FILTER(x)]</td></tr>
<tr><td>(Q2)</td><td>that contain NP[x]</td><td>→</td><td>CP[SEARCH(x)]</td></tr>
<tr><td>(Q3)</td><td>ordered by NP[x] in ascending|descending order</td><td>→</td><td>CP[ORDERBY(x, 'asc'|'desc')]</td></tr>
<tr><td>(Q4)</td><td>the NP[x] of</td><td>→</td><td>NP/NP[SELECT(x)]</td></tr>
<tr><td>(Q5)</td><td>the top NUMBER[x] of</td><td>→</td><td>NP/NP[TOP(x)]</td></tr>
<tr><td>(Q6)</td><td>the number of</td><td>→</td><td>NP/NP[COUNT())]</td></tr>
<tr><td></td><td colspan="3">[Glue]</td></tr>
<tr><td>(G1)</td><td>that VP[x]</td><td>→</td><td>CP[x]</td></tr>
<tr><td>(G2)</td><td>CP[x], and CP[y]</td><td>→</td><td>CP[x, y]</td></tr>
<tr><td>(G3)</td><td>NP/NP[x] NP/NP[y]</td><td>→</td><td>NP/NP[x, y]</td></tr>
<tr><td>(G4)</td><td>V[x] NP/NP[w] NP[y] CP[u]</td><td>→</td><td>S[x-y{w, u}]</td></tr>
</table>

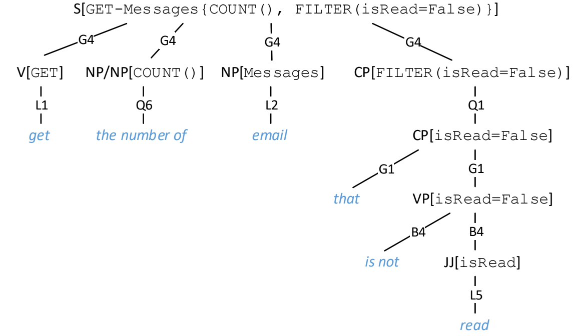


Figure 4: Canonical command generation. Left: lexicon and grammar. Right: derivation example.

3.1 API Call and Canonical Command

We generate API calls solely from the specification of an API. In addition to the schema items like query options and entity properties, we also need property values to generate API calls, which are not available in the API specification. For properties of enumerable value type, e.g., `Boolean`, we enumerate all the possible values (`True`/`False`). For properties with unconstrained value type, e.g., `Datetime`, we synthesize a few representative values for each property, e.g., `today` or `this_week` for `receivedDateTime`. Note that these are abstract values at the API frame level, and will be converted into real values according to the context (e.g., real-world datetime) when an API frame is converted into a real API call.

We can easily enumerate all the combinations of query options, properties, and property values to generate API calls. Simple heuristics can be used to reduce combinations that are not very sensible. For example, `TOP` has to be applied on a sorted list, so has to be used together with `ORDERBY`. Also, Boolean properties like `isRead` cannot be used in `ORDERBY`. But still, due to combinatorial explosion, there are still a large number of API calls for each API.

Average users cannot understand API calls. So similar to [28], we convert an API call into a canonical command. We define an API-specific *lexicon* and an API-general *grammar* (Figure 4). The lexicon supplies a *lexical form*, along with a *syntactic category*, for each item (HTTP verbs, entities, properties, and property values). For example, the lexicon entry ⟨sender → NP[`from`]⟩ specifies that the lexical form of the property `from` is "*sender*", and the syntactic category is noun phrase (NP), which will be used in the grammar. The syntactic category can also be verb (V), verb phrase (VP), adjective (JJ), complementizer phrase (CP), generalized noun phrase which is followed by another noun phrase (NP/NP), generalized prepositional phrase (PP/NP), sentence (S), etc. It is worth noting that although the lexion is specific to each API and has to be provided by the administrator of the API, the grammar is designed to be general, and can be re-used for any RESTful API following the OData protocol directly or with slight modification. The 17 grammar rules in Figure 4 can cover all the API calls used in the following experiments (Section 5).

The grammar specifies how to step by step derive a canonical command from an API call. It is a set of rules in the form $\langle t_1, t_2, ..., t_n \rightarrow c[z] \rangle$, where $\{t_i\}_{i=1}^n$ is a sequence of tokens, z is a (partial) API call, and c is its syntactic category. We talk through the example in Figure 4. For the API call at the root of the derivation tree, because its syntactic category is S, we first apply rule **G4**, which split the full API call into 4 partial API calls. According to their syntactic category, the first 3 can be directly converted into natural language phrases, while the last one takes another derivation subtree to be converted into a complementizer phrase "*that is not read*." One thing to note is that syntactic categories enable conditional derivation. For example, if we are at VP[`x = False`], both rule **B2** and rule **B4** can be applied, the syntactic category of `x` then helps make the decision. If the syntactic category of `x` is VP, rule **B2** is triggered (e.g., `x` is `hasAttachments` → "*do not have attachment*"); if it is JJ, rule **B4** is triggered (e.g., `x` is `isRead` → "*is not read*"). This avoids awkward canonical commands ("*do not read*" or "*is not have attachment*") and makes the generated canonical commands more natural.

3.2 Semantic Mesh

We can generate a lot of API calls using the above approach, but it is not economic to annotate all of them via crowdsourcing. Next we propose a hierarchical probabilistic model for the crowdsourcing process, which provides information to later decide which API calls to annotate. To the best of our knowledge, this is the first probabilistic model for the crowdsourcing process of NLIs, which is featured by the unique and intriguing challenge of modeling the interplay of natural language and formal meaning representations.

Formal meaning representations in general, and API calls in particular, are compositional by nature. For example, z_{12} = `GET-Messages{COUNT(),FILTER(isRead=False)}` is composed of z_1 = `GET-Messages{FILTER(isRead=False)}` and z_2 = `GET-Messages{COUNT()}` (we will refer to these examples again later). Our key

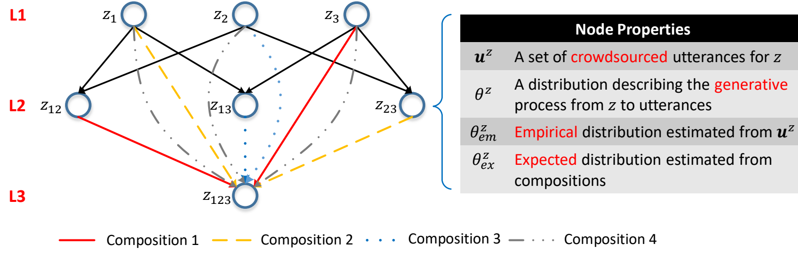


Figure 5: Semantic Mesh. The i-th layer consists of API calls with i parameters. Edges represent compositions. Probability distributions on nodes characterize the corresponding language models.

insight is that such compositionality can be exploited to model the crowdsourcing process.

We start with the definition of composition, based on the parameter set of API calls (c.f. Figure 2). Formally,

Definition 3.1 (Composition). Given an API and a set of API calls $z_1, z_2, ..., z_{n+1}, n > 1$, if we denote $r(z)$ as the parameter set of z, $\{z_1, z_2, \ldots, z_n\}$ is a *composition* of z_{n+1} iff. $\{r(z_1), r(z_2), \ldots, r(z_n)\}$ is a partition of $r(z_{n+1})$.

Based on the composition relationship of API calls, we can organize all the API calls of an API into a hierarchical structure. API calls with the same number of parameters are represented as nodes residing in the same layer, and compositions are represented as directed edges between layers. We call this structure *semantic mesh* (or SeMesh for short), illustrated in Figure 5.

Similar to the language modeling approach in information retrieval [23], we assume the utterances corresponding to the same API call z are generated via a stochastic process, characterized by a language model θ^z. For simplicity, we will focus on word probabilities, thus $\theta^z: p(w|z), w \in \mathcal{V}$, where $\mathcal{V}$ is the vocabulary. For reasons that will become clear soon, instead of using the standard unigram language model, we propose to use a Bag of Bernoulli (BoB) distribution. Each Bernoulli distribution corresponds to a random variable W of *whether word w appears in an utterance generated from z,* and the BoB distribution is a bag of the Bernoulli distribution of all the words, $\{p(W|z)\}$. We will use $p_b(w|z)$ as a shorthand for $p(W|z)$.

Suppose we have collected a (multi-)set of utterances $\boldsymbol{u}^z$ for z, the maximum likelihood estimation (MLE) of the BoB distribution is the fraction of utterances containing w:

$$p_b(w|z) = \frac{|\{u|w \in u, u \in \boldsymbol{u}^z\}|}{|\boldsymbol{u}^z|}, \forall w \in \mathcal{V}. \quad (1)$$

Example 2. Given the aforementioned API call z_1, suppose we have collected two utterances u_1 = "*find unread emails*" and u_2 = "*emails that are not read*," then $\boldsymbol{u}^z = \{u_1, u_2\}$. p_b("*emails*"$|z) = 1.0$ because word "*emails*" appears in both utterances. Similarly, p_b("*unread*"$|z) = 0.5$ and p_b("*meeting*"$|z) = 0.0$.

There are three basic node-level operations in semantic mesh: annotate, compose, and interpolate.

ANNOTATE is to collect utterances $\boldsymbol{u}^z$ that paraphrase the canonical command of a node z via crowdsourcing, and estimate an empirical distribution θ^z_{em} using maximum likelihood estimation.

COMPOSE tries to infer a language model based on the compositions of the node, leading to the *expected distribution* θ^z_{ex}. Suppose $\{z_1, z_2, \ldots, z_n\}$ is a composition of z, if we assume their corresponding utterances follow this composition relationship, then θ^z_{ex} should *factorize* over $\{\theta^{z_1}, \theta^{z_2}, ..., \theta^{z_n}\}$:

$$\theta^z_{ex} = f(\theta^{z_1}, \theta^{z_2}, ..., \theta^{z_n}), \quad (2)$$

where f is a composition function. For the BoB distribution, the composition function will be:

$$p_b(w|z) = 1 - \prod_{i=1}^{n}(1 - p_b(w|z_i)). \quad (3)$$

In other words, suppose u_i is an utterance of z_i, u an utterance of z, if $\{u_i\}_{i=1}^n$ compositionally form u, then word w is not in u iff. it is not in any u_i. When z has multiple compositions, θ^z_{ex} is computed separately and then averaged. The standard unigram language model does not lead to a natural composition function. The normalization of word probabilities involves the length of utterances, which in turn involves the complexity of API calls, breaking the factorization in Eq. (2). It motivates us to propose the BoB distribution.

Example 3. Suppose we have annotated the aforementioned API call z_1 and z_2, each with two utterance, $\boldsymbol{u}^{z_1}$ = {"*find unread emails*", "*emails that are not read*"}, and $\boldsymbol{u}^{z_2}$ = {"*how many emails do I have*", "*find the number of emails*"}. We have estimated the language model θ^{z_1} and θ^{z_2}. The compose operation tries to estimate $\theta^{z_{12}}_{ex}$ without requiring $\boldsymbol{u}^{z_{12}}$. For example, for the word "*emails*", p_b("*emails*"$|z_1) = 1.0$, and p_b("*emails*"$|z_2) = 1.0$, thus according to Eq. (3), p_b("*emails*"$|z_{12}) = 1.0$, meaning that we believe this word will appear in any utterance of z_{12}. Similarly, p_b("*find*"$|z_1) = 0.5$, and p_b("*find*"$|z_2) = 0.5$, thus p_b("*find*"$|z_{12}) = 0.75$. Because the word has a good chance to be generated from either z_1 or z_2, its probability for z_{12} should be higher.

Of course, utterances are not always combined compositionally. For example, multiple items in a formal meaning representation can be compressed into a single word or phrase in natural language, a phenomenon coined as *sublexical compositionality* [28]. One such example is shown in Figure 3, where three parameters, `TOP(1)`, `FILTER(start>now)`, and `ORDERBY(start,asc)`, are compressed into a single word "*next*". However, it is impossible to get such information without annotating an API call, reminiscent of the chicken-and-egg problem. In the absence of such information, it is reasonable to adopt a default assumption that *utterances follow the composition relationship of API calls*, which makes it possible to compute the expected distribution. As we will experimentally show in Section 5.3, this is a plausible assumption. It is worth noting that

this assumption is only used to model the crowdsourcing process for data collection. During testing time, utterances issued by real users can violate this assumption. It is possible for the natural language interface to handle such non-compositional cases if they are covered by the collected training data.

INTERPOLATE combines all the available information about z, i.e., the annotated utterances of z and the information inherited from compositions, and get a more accurate estimation of θ^z by interpolating θ^z_{em} and θ^z_{ex}.

$$\theta^z = \alpha * \theta^z_{em} + (1 - \alpha) * \theta^z_{ex}, 0 \leq \alpha \leq 1. \tag{4}$$

The balance parameter α controls the trade-off between the annotations of the current node, which are accurate but scarce, and the information inherited from compositions based on the compositionality assumption, which may not be as accurate but is rich. In some sense, θ^z_{ex} serves a similar purpose as smoothing in language modeling [33], which is to better estimate the probability distribution when there is insufficient data (annotations). A larger α means a larger weight on θ^z_{em}. For a root node that has no composition, $\theta^z = \theta^z_{em}$. For an unannotated node, $\theta^z = \theta^z_{ex}$.

Now we describe an algorithm to update a semantic mesh, i.e., to compute θ^z for all z (Algorithm 1), even when only a small portion of nodes have been annotated. We assume θ^z_{em} is already up-to-date for all annotated nodes. In a top-down, layer-wise manner, we consecutively compute θ^z_{ex} and θ^z for every node z. Upper layers much be updated first so that the expected distribution of lower-layer nodes can be computed. As long as we have annotated all the root nodes, we can compute θ^z for all the nodes.

Algorithm 1 Update Node Distributions of Semantic Mesh

```
1: function SeMesh.update( )
2:     for all layer from top to bottom do
3:         for all node z in the current layer do
4:             z.compose()
5:             z.interpolate()
```

3.3 Crowdsourcing Optimization

Semantic mesh gives a holistic view of the whole API call space as well as the interplay of utterances and API calls, baaed on which we can selectively annotate only a subset of high-value API calls. In this section, we propose a differential propagation strategy for crowdsourcing optimization.

Given a semantic mesh, suppose the node set is Z, our goal is to iteratively select a subset of nodes $\bar{Z} \subset Z$ for crowd workers to annotate. If we call the set of nodes annotated so far as the *state*, then what we will be seeking for is a *policy* $\pi \colon Z \setminus \bar{Z} \to \mathbb{R}$ that scores each unannotated node based on the current state.

Before plunging into discussing how to compute a good policy, let's assume we already have that, so we can first sketch our crowdsourcing algorithm at a high level (Algorithm 2), and discuss related techniques. More specifically, we first annotate all root nodes so that we can estimate the distribution for all the nodes in Z (line 3). In each iteration, we update node distributions (line 5), compute the policy based on the current state of the semantic mesh (line 6), greedily select the unannotated node with the highest score (line 7), and annotate the node and result in a new state (line 8). In practice, one can also annotate multiple nodes per iteration for efficiency.

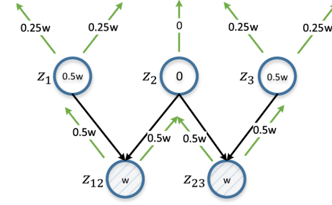


Figure 6: Differential propagation. z_{12} and z_{23} are the node pair under examination. w is a score computed based on $d(z_{12}, z_{23})$, and is propagated iteratively from the bottom up, halved in each iteration. A node's score is the absolute difference of its scores from z_{12} and z_{23} (thus *differential*). z_2 gets a 0 score because it is a shared parent of z_{12} and z_{23}; annotating it will not help much in distinguishing z_{12} and z_{23}.

Under a broad definition, our problem can be categorized as an active learning problem [22], with the same goal to select a subset of examples to annotate, in order to get a training set that can improve the performance of a learner. However, several key distinctions make classic active learning techniques like uncertainty sampling [14] not directly applicable. In a typical active learning setting, a learner, which would be an NLI in our case, tries to learn a mapping $f : X \to Y$, where X is the input example space, consisting of a small set of labeled examples and a large set of unlabeled examples, and Y is usually a small set of class labels. The learner evaluates the *informativeness* of the unlabeled examples, and selects the most informative one to get a label in Y from crowd workers. In our problem, however, the annotation task is the other way around. We need to select an instance from Y, a large API call space, and ask crowd workers to label it with examples in X, the utterance space. We also do not assume to be tied with a specific learner. Therefore, we propose a new solution for our problem. Nonetheless, we take inspiration from the rich literature of active learning.

We first define an *objective*, based on which we quantify the *informativeness* of nodes. Intuitively, we want different API calls to be *distinguishable*. In semantic mesh, it means that the distribution θ^z of different nodes are divergent enough. We first represent each θ^z as an n-dimensional vector, $(p_b(w_1|z), \ldots, p_b(w_n|z))$, where $n = |\mathcal{V}|$ is the vocabulary size. Under a certain vector distance metric d (we use $L1$ distance between the vectors in the experiments), we denote $d(z_i, z_j) = d(\theta^{z_i}, \theta^{z_j})$, i.e., the distance of two nodes is that of their distributions. A straightforward objective is then to maximize the sum of the distance between all pairs of nodes. However, optimizing over all pair-wise distances could be computationally prohibitive and also unnecessary. A distant node pair can already be easily distinguished, so it is less beneficial to further increase their distance. Instead, we can focus on the node pairs that are causing the most confusion, i.e., the ones with the smallest distance.

$$\Theta = \sum_{i=1}^{K} d(z_i, z'_i), \tag{5}$$

where $\{(z_1, z'_1), \ldots, (z_K, z'_K)\}$ are the first K node pairs if we rank all node pairs by distance in ascending order.

Algorithm 2 Iteratively Annotate a Semantic Mesh with a Policy

Input: number of nodes to annotate T (budget)

```
1: function SeMesh.annotate( )
2:     for all root node z do
3:         z.annotate()
4:     for i ← 1 to T − |root nodes| do
5:         SeMesh.update()
6:         π ← SeMesh.compute_policy()
7:         z ← arg max_z π(z)
8:         z.annotate()
```

Algorithm 3 Compute Policy based on Differential Propagation

Input: number of node pairs K

```
1: function SeMesh.compute_policy( )
2:     compute pair-wise distance of all nodes
3:     {(z_1, z'_1), ..., (z_K, z'_K)} ← K closest pairs
4:     π ← {z: 0} for unannotated z
5:     for i ← 1 to K do
6:         w ← min(1.0, 1/d(z_i, z'_i))
7:         scores_1 ← {z: 0} for all z
8:         scores_2 ← {z: 0} for all z
9:         SeMesh.propagate(z_i, w, scores_1)
10:        SeMesh.propagate(z'_i, w, scores_2)
11:        for all unannotated node z do
12:            π(z) ← π(z) + |scores_1(z) − scores_2(z)|
13:    return π
```

Algorithm 4 Recursively Propagate a Score from a Source Node to All Its Parent Nodes

Input: source node z, initial score w, current score dictionary *scores* (passed by reference)

```
1: function SeMesh.propagate(z, w, scores)
2:     scores(z) ← scores(z) + w
3:     if z is not root node then
4:         w ← w/2                          ▷ halved in each iteration
5:         for all composition of z do
6:             for all node z' in current composition do
7:                 SeMesh.propagate(z', w, scores)
```

A node is more informative if annotating it can potentially increase the objective Θ more. We propose a *differential propagation* strategy to quantify this. For a node pair whose distance is small, we examine all their parent nodes: If it is shared by the node pair, it should get a low score because annotating it will change both nodes similarly; Otherwise, it should get a high score, and the closer the node pair is, the higher the score should be. For example, if the distance of the nodes, "*unread emails about PhD application*" and "*how many emails are about PhD application*", are small, then annotating their shared parent node, "*emails about PhD application*", will not help much in distinguishing the two nodes; rather, we shall annotate their parents nodes that are not shared, like "*unread emails*" and "*how many emails*". An illustrative example is shown in Figure 6, and the algorithm is outlined in Algorithm 3. We take the reciprocal of node distance, capped by a constant, as the score (line 6), so that closer node pairs make a larger impact. For a node pair, we in parallel propagate the score from each node to all its parent nodes (line 9, 10 and Algorithm 4). The score of an unannotated node is the absolute difference of its scores from a node pair, accumulated across all the node pairs (line 12).

4 NATURAL LANGUAGE INTERFACE

To evaluate the proposed framework, we need to train NL2API models using the collected data. There is yet any NL2API model readily available, but we adapt two successful NLI models from other domains to APIs.

4.1 LM based Retrieval Model

Following recent development of NLI to knowledge bases [5, 31], we can treat NL2API as a retrieval problem, and adapt the language model based retrieval model (LM, [23]) to our setting.

Given an input utterance u, the task is to find the API call z in the semantic mesh that best matches u. We first convert the BoB distribution $\theta^z = \{p_b(w_i|z)\}$ of each API call z into a unigram language model:

$$p_{lm}(w_i|z) = \frac{p_b(w_i|z) + \beta}{\sum_i^{|\mathcal{V}|} p_b(w_i|z) + \beta|\mathcal{V}|}, \tag{6}$$

where we use additive smoothing, and $0 \le \beta \le 1$ is a smoothing parameter. A larger β means more weight to unseen words. API calls can then be ranked by their log-likelihood:

$$\begin{aligned} \log p(z|u) &\propto \log p(u|z) + \log p(z) \qquad \text{(assuming uniform prior)} \\ &\propto \sum_i^{|u|} \log p_{lm}(w_i|z) \end{aligned} \tag{7}$$

The highest-ranked API call is then used as the model output.

4.2 Seq2Seq Paraphrase Model

Neural networks are becoming a popular model choice for NLIs [12, 31], among which the Seq2Seq [27] model is in particular suitable for NLIs, because it can naturally handle input and output sequences of variable length. Here we adapt it for NL2API.

For an input sequence $\boldsymbol{x} = (x_1, x_2, \dots, x_T)$, the model estimates the conditional probability distribution $p(\boldsymbol{y}|\boldsymbol{x})$ for all possible output sequences $\boldsymbol{y} = (y_1, y_2, \dots, y_{T'})$. The lengths T and T' can be different, and both of them can be varied. In NL2API, $\boldsymbol{x}$ is an input utterance. $\boldsymbol{y}$ could be either a serialized API call, or its canonical command. We will use canonical commands as the target output sequences, which in fact turns it into a paraphrasing problem [5].

An *encoder*, which is implemented as a recurrent neural network (RNN) with gated recurrent units (GRUs) [8], first encodes $\boldsymbol{x}$ into a fixed-dimensional vector,

$$\boldsymbol{h}_0 = RNN(\boldsymbol{x}), \tag{8}$$

where RNN is a shorthand for applying a GRU on the whole input sequence token by token and output its last hidden state.

The *decoder*, also implemented as an RNN with GRUs, takes $\boldsymbol{h}_0$ as its initial state, and processes the output sequence $\boldsymbol{y}$ one token at a time to generate a sequence of states,

$$\boldsymbol{h}_t = GRU(y_t, \boldsymbol{h}_{t-1}). \tag{9}$$

The *output* layer takes each decoder state as input and generates a distribution over vocabulary $\mathcal{V}$ as output. We simply use an affine transformation followed by softmax,

$$\begin{aligned} \boldsymbol{o}_t &= \{p(o_t = w) | w \in \mathcal{V}\} \\ &= softmax(\boldsymbol{W}_o \boldsymbol{h}_t + \boldsymbol{b}_o). \end{aligned} \tag{10}$$

The final conditional probability, which measures how well an canonical command $\boldsymbol{y}$ paraphrases the input utterance $\boldsymbol{x}$, is $p(\boldsymbol{y}|\boldsymbol{x}) = \prod_{t=1}^{T'} p(o_t = y_t)$. API calls are then ranked by the conditional probability of their canonical command. We refer readers to [27] for more details of how to train the model.

5 EXPERIMENTS

We experimentally investigate the following research questions: [**RQ1**]: *Can we collect high-quality training data through the proposed framework at a reasonably low cost?* [**RQ2**]: *Does semantic mesh provide a more accurate estimation of language models than maximum likelihood estimation?* [**RQ3**]: *Does the differential propagation strategy make the crowdsourcing process more effective?*

5.1 Crowdsourcing

We apply the proposed framework to two web APIs from Microsoft, `GET-Events` and `GET-Messages`, which provide advanced search services for a user's emails and calendar events, respectively. We build a semantic mesh for each API by enumerating all API calls (Section 3.1) with up to 4 parameters. The API call distribution is shown in Table 2. We use an internal crowdsourcing platform similar to Amazon Mechanical Turk. For the agility of experimentation, we have annotated all API calls with up to 3 parameters. However, we will only use a certain subset for training in each specific experiment. Each API call is annotated with 10 utterances, and we pay 10 cents for each utterance. There are 201 participants in the annotation task, who have been filtered using a qualification test. The workers took 44 seconds on average to paraphrase a canonical command, so we can get 82 training examples from a worker per hour with a cost of 8.2 dollars, which we believe is reasonably low. In terms of annotation quality, we manually examined 400 collected utterances, and found the error rate to be 17.4%. The main causes of error are missing certain parameters (e.g., forgetting a `ORDERBY` or a `COUNT` parameter) or misunderstanding certain parameters (e.g., ranking by ascending order when it is in fact by descending order), each of which accounting for roughly half of the errors. The error rate is on par with other crowdsourcing efforts for NLIs [28]. In summary, we posit that the answer to [**RQ1**] is positive. The data quality can be further improved by employing independent crowd workers for post-verification.

In addition, we have annotated an *independent* testing set randomly selected from the whole semantic mesh, including API calls with 4 parameters (Table 3). Each API call in the testing set was initially annotated with 3 utterances. We then screened and discarded the erroneous ones to ensure the testing quality. Finally the testing set contains 61 API calls and 157 utterances for `GET-Messages`, and 77 API calls and 190 utterances for `GET-Events`. All the testing utterances are not included in training, and there are even many testing API calls (e.g., those with 4 parameters) that are never seen in training, making it a very challenging testing set.

Table 2: API call distribution.

# of Params	1	2	3	4
`GET-Messages`	16	101	363	872
`GET-Events`	16	117	528	1679

Table 3: Testing set distribution: utterances (calls).

# of Params	1	2	3	4
`GET-Messages`	8 (3)	35 (14)	62 (24)	52 (20)
`GET-Events`	21 (7)	35 (15)	61 (25)	73 (30)

5.2 Experiment Setup

We use accuracy, i.e., the proportion of testing utterances for which the top prediction is correct, as the evaluation metric. If not otherwise stated, the balance parameter $\alpha = 0.3$, and the smoothing parameter of LM $\beta = 0.001$. The number of node pairs K used in differential propogation is set to 100 thousand. For the Seq2Seq model, the state size of both the encoder and the decoder are set to 500. The parameters are selected based on a preliminary study on a separate validation set (independent from testing).

In addition to its usefulness for crowdsourcing optimization, as the first-of-its-kind model for the crowdsourcing process of NLI, semantic mesh bears its own technical merits. So we will evaluate semantic mesh and the optimization algorithm separately.

5.3 Semantic Mesh Evaluation

Overall Performance. In this experiment, we evaluate the semantic mesh model, and in particular, whether the operations COMPOSE and INTERPOLATE lead to better estimation of language models. The quality of language models can be reflected by the performance of the LM model: the more accurate estimation, the better performance. We use several training sets, corresponding to different sets of annotated nodes. ROOT is all the root nodes. TOP2 is ROOT plus all the nodes in layer 2, and TOP3 is TOP2 plus all the nodes in layer 3. This allows us to evaluate semantic mesh with different amount of training data.

The results are shown in Table 4. For the baseline LM model, we use maximum likelihood estimation (MLE) for language model estimation, i.e., we use θ_{em}^z for all annotated nodes, and the uniform distribution for unannotated nodes. Not surprisingly, the performance is rather poor, especially when the number of annotated nodes is small, because MLE cannot provide any information about the unannotated nodes. When we add COMPOSE to MLE, we are able to estimate the expected distribution θ_{ex} for unannotated nodes, but θ_{em} is still used for the annotated nodes, i.e., no INTERPOLATE. This brings a significant improvement across APIs and training sets. *With only 16 annotated API calls (ROOT), the simple LM model with SEMESH can outperform the more advanced Seq2Seq model with more than 100 annotated API calls (TOP2), and close to its performance when using around 500 annotated API calls (TOP3).* These results clearly show that the language models estimated by the COMPOSE operation is quite accurate, and empirically demonstrate the plausibility of the compositionality assumption of utterances (Section 3.2). It can be observed that `GET-Events` is in general

Table 4: Overall accuracy in percentage. Semantic mesh operations significantly improve the simple LM model, making it outperform the more advanced Seq2Seq model when there is only a moderate amount of training data. The results demonstrate that semantic mesh provides accurate estimation of language models.

		ROOT	TOP2	TOP3
GET-Messages	LM	3.18	12.1	25.5
	LM + COMPOSE	**46.5**	40.8	36.3
	LM + COMPOSE + INTERPOLATE	**46.5**	**47.8**	50.3
	Seq2Seq	5.73	22.9	**57.3**
GET-Events	LM	7.89	13.7	17.4
	LM + COMPOSE	**39.0**	37.9	31.6
	LM + COMPOSE + INTERPOLATE	**39.0**	**42.6**	44.2
	Seq2Seq	8.95	20.5	**45.3**

harder than GET-Messages. This is because GET-Events involves more temporal commands that are challenging; events can be either in the future or past, while emails are always from the past.

The performance of LM + COMPOSE decreases when we use more training data, which shows the inadequacy of solely using θ_{em}, and the need of combining θ_{em} with θ_{ex}. When we INTERPOLATE θ_{em} and θ_{ex}, we get further improvements except on ROOT, where none of the nodes has both θ_{em} and θ_{ex}. In contrast with COMPOSE, the more training data we use, the more significant improvement INTERPOLATE brings. Overall, semantic mesh brings a significant improvement over the MLE baseline. We posit that the answer to [**RQ2**] is positive.

The best accuracies are in the range of 0.45 to 0.6, which are not very high but are on par with the state-of-the-art methods on NLI to knowledge bases [31]. This reflects the difficulty of the problem, as the model has to accurately find the best API call from thousands of related API calls. Collecting more utterances for each API call (c.f. Figure 7) and using more advanced models, such as bi-directional RNN with attention mechanism [25], may further improve the performance, which we leave for future work.

Impact of Hyper-parameters. We now examine the impact of the two hyper-parameters in semantic mesh, the number of utterances $|\boldsymbol{u}|$ and the balance parameter α, still based on the performance of the LM model (Figure 7). Utterances are randomly sampled when $|\boldsymbol{u}| < 10$, and mean scores from 10 repeated runs are reported. We only show the results of GET-Events, and the results of GET-Messages are similar.

Not surprisingly, the more utterances we annotate for each node, the better performance we get, although the gain gradually shrinks. So it is often a good idea to get more utterances if one can afford. On the other hand, the model's performance is not very sensitive to α, as long as it is in a reasonable range ($[0.1, 0.7]$). The impact of α increases when there are more annotated nodes, which is expected, because interpolation only influences annotated nodes.

5.4 Crowdsourcing Optimization

In this experiment, we evaluate the proposed differential propagation (DP) strategy for crowdsourcing optimization. Different crowdsourcing strategies iteratively select API calls to annotate. In each iteration 50 API calls are selected by each strategy and then

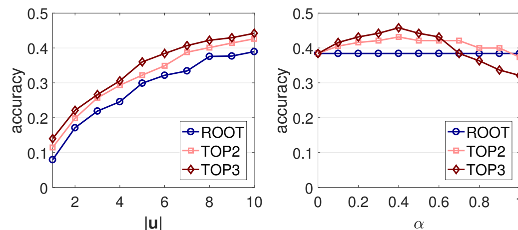


Figure 7: Impact of hyper-parameters.

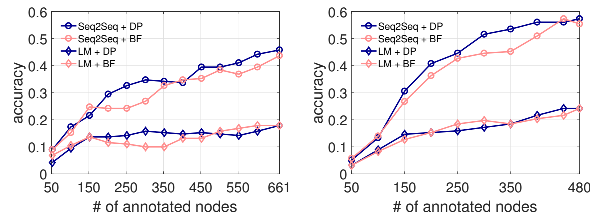


Figure 8: Crowdsourcing optimization experiment. Left: GET-Events. Right: GET-Messages.

annotated, and the two NL2API models will be trained on the accumulated annotated data, and then get evaluated on the testing set. We use the basic LM model, which is independent of semantic mesh. A better crowdsourcing strategy should lead to better model performance for the same amount of annotations. Instead of annotating the nodes on the fly via crowdsourcing, we use the annotations we have already collected as the candidate pool (Section 5.1), so all strategies will only select from API calls with up to 3 parameters.

The baseline strategy is *breadth first* (BF), which, in a top-down fashion, gradually annotates each layer of the semantic mesh. This mimics the strategy used in [28]. It is a strong baseline. The upper-layer API calls are usually more important, because they compose the lower-layer API calls.

The experiment results are shown in Figure 8. For both NL2API models and both APIs, DP in general leads to better performance. When we only annotate 300 API calls for each API, for the Seq2Seq model, DP brings an absolute accuracy gain over 7% on both APIs. When it comes close to exhaust the candidate pool, the two algorithms converge, which is expected. The results show that DP is able to identify the API calls with high value for training an NL2API. In summary, we posit that the answer to [**RQ3**] is positive.

6 RELATED WORK

Natural Language Interface. Research on natural language interface (NLI) can date back to decades ago [29]. Early NLIs are mostly rule-based. In the past years, learning-based methods have become the mainstream. Popular learning algorithms include log-linear models [4, 32] and more recently deep neural networks [26, 31]. There has been considerable research on NLI for relational databases [32], knowledge bases [4], and web tables [26], but little for APIs. Two main hurdles for NL2API are the lack of a unified semantic representation for APIs, and, partly due to that, the lack of training data. We address both of them. We propose a unified semantic representation for APIs that follow the REST style [1], and a novel approach to collect training data in this representation.

Training Data Collection for NLI. Training data for NLIs has conventionally been collected in a best-effort manner. For example, Berant et al. [4] collect NL questions through the Google Suggest API, while Quirk et al. [19] collect NL commands and the corresponding API calls from the IFTTT website. Only lately researchers start to study how to build an NLI by collecting training data via crowdsourcing. While crowdsourcing has become a common practice in language-related research, it is particularly challenging and intriguing when it comes to NLI, because of the complicated interplay of natural language and formal meaning representations. Most existing work in this direction is from NLI to knowledge bases [6, 24, 28], where the formal representation is logical forms in a certain logical formalism. Wang et al. propose the idea of converting logical forms into canonical commands using a grammar [28], while in [24] techniques are developed to refine the generated logical forms and filter out the ones that do not correspond to a meaningful NL question. A semi-automated framework is proposed in [13] to interact with users on the fly to map NL commands into smartphone API calls. Similarly, Huang at al. [11] propose a crowdsourcing framework where workers participate in an interactive dialog to fulfill the annotation task for web API. But none of the existing work has exploited the compositionality of formal representations to optimize the crowdsourcing process.

Semantic Techniques for Web API. Some other semantic techniques have emerged for web APIs. For example, semantic descriptions of web APIs are extracted in [17] to facilitate API composition, while in [20] a search mechanism is proposed to help find web APIs for composition. NL2API can potentially be applied to such problems too, e.g., as a unified search engine for finding APIs.

7 CONCLUSIONS AND FUTURE WORK

We formulated the natural language interface to web API (NL2API) problem, and proposed an end-to-end framework to build an NL2API from scratch, with the novel technical contributions in collecting training data via crowdsourcing. It opens up an array of future directions: (1) *Language model.* How to generalize from single words to more complex language units like phrases? (2) *Crowdsourcing optimization.* How to better exploit the semantic mesh? (3) *NL2API model.* For example, the slot filling framework for spoken dialog systems could be a good fit to our API frame representation. (4) *API composition.* How to collect training data when multiple APIs are mashed up? (5) *Tuning from interaction*: how to continue improving an NL2API through user interaction after initial training?

REFERENCES

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004. Web services. In *Web Services.* Springer, 123–149.
[2] Collin F Baker, Charles J Fillmore, and John B Lowe. 1998. The berkeley framenet project. In *ACL.*
[3] Jerome R Bellegarda. 2014. Spoken language understanding for natural interaction: The Siri experience. In *Natural Interaction with Robots, Knowbots and Smartphones.* Springer, 3–14.
[4] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *EMNLP.*
[5] Jonathan Berant and Percy Liang. 2014. Semantic Parsing via Paraphrasing. In *ACL.*
[6] Antoine Bordes, Nicolas Usunier, Sumit Chopra, and Jason Weston. 2015. Large-scale simple question answering with memory networks. *arXiv preprint arXiv:1506.02075* (2015).
[7] Athman Bouguettaya, Quan Z Sheng, and Florian Daniel. 2014. *Web services foundations.* Springer.
[8] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv:1406.1078 [cs.CL]* (2014).
[9] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. 2010. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE transactions on Services Computing* 3, 3 (2010), 223–235.
[10] Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD.*
[11] Ting-Hao Kenneth Huang, Walter S Lasecki, and Jeffrey P Bigham. 2015. Guardian: A Crowd-Powered Spoken Dialog System for Web APIs. In *Third AAAI Conference on Human Computation and Crowdsourcing.*
[12] Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *ACL.*
[13] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services.*
[14] David D Lewis and Jason Catlett. 1994. Heterogeneous uncertainty sampling for supervised learning. In *ICML.*
[15] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *VLDB* (2014).
[16] OData. 2014. OData specification v4. http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part1-protocol.html. (2014).
[17] Luca Panziera, Marco Comerio, Matteo Palmonari, Flavio De Paoli, and Carlo Batini. 2012. Quality-driven extraction, fusion and matchmaking of semantic web api descriptions. *Journal of Web Engineering* 11, 3 (2012), 247.
[18] Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *ACL.*
[19] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL.*
[20] Ajith Ranabahu, Meenakshi Nagarajan, Amit P Sheth, and Kunal Verma. 2008. A faceted classification based approach to search and rank web apis. In *IEEE International Conference on Web Services.*
[21] Ruhi Sarikaya, Paul Crook, Alex Marin, Minwoo Jeong, Jean-Philippe Robichaud, Asli Celikyilmaz, Young-Bum Kim, Alexandre Rochette, Omar Zia Khan, Xiuahu Liu, et al. 2016. An overview of end-to-end language understanding and dialog management for personal digital assistants. In *IEEE Workshop on Spoken Language Technology.*
[22] Burr Settles. 2009. *Active Learning Literature Survey.* Computer Sciences Technical Report 1648. University of Wisconsin–Madison.
[23] Fei Song and W Bruce Croft. 1999. A general language model for information retrieval. In *CIKM.*
[24] Yu Su, Huan Sun, Brian Sadler, Mudhakar Srivatsa, Izzeddin Gür, Zenghui Yan, and Xifeng Yan. 2016. On Generating Characteristic-rich Question Sets for QA Evaluation. In *EMNLP.*
[25] Yu Su and Xifeng Yan. 2017. Cross-domain Semantic Parsing via Paraphrasing. In *EMNLP.*
[26] Huan Sun, Hao Ma, Xiaodong He, Wen-tau Yih, Yu Su, and Xifeng Yan. 2016. Table cell search for question answering. In *WWW.*
[27] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *NIPS.* 3104–3112.
[28] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a Semantic Parser Overnight. In *ACL.*
[29] William A Woods. 1973. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the American Federation of Information Processing Societies Conference.*
[30] Mohamed Yahya, Klaus Berberich, Shady Elbassuoni, Maya Ramanath, Volker Tresp, and Gerhard Weikum. 2012. Deep answers for naturally asked questions on the web of data. In *WWW.*
[31] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *ACL.*
[32] Luke Zettlemoyer and Michael Collins. 2005. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. In *UAI.*
[33] Chengxiang Zhai and John Lafferty. 2004. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems* 22, 2 (2004), 179–214.
[34] Li Zhang, Chris Stover, Amanda Lins, Chris Buckley, and Prasant Mohapatra. 2014. Characterizing mobile open apis in smartphone apps. In *Networking Conference, 2014 IFIP.* IEEE, 1–9.
[35] Wancai Zhang, Hailong Sun, Xudong Liu, and Xiaohui Guo. 2014. Temporal QoS-aware web service recommendation via non-negative tensor factorization. In *WWW.*